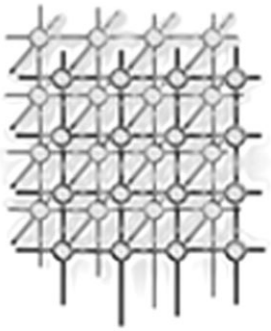
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2006; **18**:1039–1065
Published online 13 December 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.994

Scientific workflow management and the Kepler system



Bertram Ludäscher[1,2,*,†], Ilkay Altintas[1], Chad Berkley[3], Dan Higgins[3], Efrat Jaeger[1], Matthew Jones[3], Edward A. Lee[4], Jing Tao[1] and Yang Zhao[4]

[1]*San Diego Supercomputer Center, UC San Diego, San Diego, CA 92093, U.S.A.*
[2]*Department of Computer Science and Genome Center, UC Davis, Davis, CA 95616, U.S.A.*
[3]*National Center for Ecological Analysis and Synthesis, UC Santa Barbara, Santa Barbara, CA 93101, U.S.A.*
[4]*Department of Electrical Engineering and Computer Sciences, UC Berkeley, Berkeley, CA 94720, U.S.A.*

SUMMARY

Many scientific disciplines are now data and information driven, and new scientific knowledge is often gained by scientists putting together data analysis and knowledge discovery 'pipelines'. A related trend is that more and more scientific communities realize the benefits of sharing their data and computational services, and are thus contributing to a distributed data and computational community infrastructure (a.k.a. 'the Grid'). However, this infrastructure is only a means to an end and ideally scientists should not be too concerned with its existence. The goal is for scientists to focus on development and use of what we call *scientific workflows*. These are networks of analytical steps that may involve, e.g., database access and querying steps, data analysis and mining steps, and many other steps including computationally intensive jobs on high-performance cluster computers. In this paper we describe characteristics of and requirements for scientific workflows as identified in a number of our application projects. We then elaborate on Kepler, a particular scientific workflow system, currently under development across a number of scientific data management projects. We describe some key features of Kepler and its underlying Ptolemy II system, planned extensions, and areas of future research. Kepler is a community-driven, open source project, and we always welcome related projects and new contributors to join. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: scientific workflows; Grid workflows; scientific data management; problem-solving environments; dataflow networks

*Correspondence to: Bertram Ludäscher, Department of Computer Science and Genome Center, UC Davis, Davis, CA 95616, U.S.A.
†E-mail: ludaesch@ucdavis.edu

Contract/grant sponsor: NSF/ITR; contract/grant numbers: 0225676 (SEEK); CCR-00225610 (Chess); 0225673 (GEON); 0325963 (ROADNet)
Contract/grant sponsor: DOE SciDAC; contract/grant number: DE-FC02-01ER25486 (SDM)
Contract/grant sponsor: NIH/NCRR; contract/grant number: 1R24 RR019701-01
Contract/grant sponsor: Biomedical Informatics Research Network Coordinating Center (BIRN-CC)
Contract/grant sponsor: NSF/DBI; contract/grant number: 0078296 (Resurgence)

Received 1 June 2004
Revised 6 April 2005
Accepted 27 April 2005

> The diversity of the phenomena of nature is so great, and the treasures hidden in the heavens so rich, precisely in order that the human mind shall never be lacking in fresh nourishment.
>
> Johannes Kepler, Mysterium Cosmographicum

1. INTRODUCTION

Information technology is revolutionizing the way many sciences are conducted, as witnessed by new techniques, results, and discoveries from quickly evolving, multi-disciplinary fields such as bioinformatics, biomedical informatics, cheminformatics, ecoinformatics, geoinformatics, etc. To further advance this new data- and information-driven science through advanced IT infrastructure, large investments are being made, e.g. in the U.K. e-Science programme, or in the U.S. through the NSF Cyberinfrastructure initiative and other initiatives from NIH (BIRN: Biomedical Informatics Research Network) and DOE (SciDAC: Scientific Discovery through Advanced Computing, GTL: Genomes to Life), just to mention a few. While many efforts focus on the underlying middleware infrastructure, known as 'the Grid', scientists are ultimately interested in tools that bring the power of distributed databases and other computational Grid resources to the desktop, and allow them to conveniently put together and run their own *scientific workflows*. By these we mean process networks that are typically used as 'data analysis pipelines' or for comparing observed and predicted data, and that can include a wide range of components, e.g. for querying databases, for data transformation and data mining steps, for execution of simulation codes on high-performance computers, etc. Ideally, the scientist should be able to plug in almost any scientific data resource and computational service into a scientific workflow, inspect and visualize data on the fly as they are computed, make parameter changes when necessary and re-run only the affected 'downstream' components, and capture sufficient metadata in the final products such that the runs of a scientific workflow, when considered as (computational) experiments themselves, help explain the results and make them reproducible by the computational scientist and others. Thus, a scientific workflow system becomes a scientific problem-solving environment, tuned to an increasingly distributed and service-oriented Grid infrastructure.

However, before this grand vision can become reality, a number of significant challenges have to be addressed. For example, current Grid software is still too complex to use for the average scientist, and fast changing versions and evolving standards require that these details be hidden from the user by the scientific workflow system. Web services seem to provide a simple basis for loosely coupled, distributed systems, but core Web service standards such as WSDL [1] only provide simple solutions to simple problems[‡], while harder problems such as Web service orchestration, third-party transfer (from one service directly to another, circumventing the transfer back to a workflow control engine), and transactional semantics of service-based workflows remain the subject of emerging or future Web service standards. The complexity of the underlying technical issues and the resulting

[‡]For example, WSDL mainly provides an XML notation for function signatures, i.e. the types of inputs and outputs of Web services.

(sometimes overly) complex standards make it less likely that those will be as widely adopted as the core standards such as XML and WSDL.

Another set of challenges arises from the inherent complexity of scientific data. For example, how can we capture more of the semantics of scientific data (beyond simple metadata meant for human consumption) and thus inform the system which data sets might be suitable input for a specific analytical pipeline? Similarly, how can we define when it is even potentially meaningful at the conceptual level to compose two independently designed Web services, or when an analysis pipeline might be included as a subworkflow in another scientific workflow? Knowledge representation techniques, including formal ontologies, and corresponding semantic Web standards such as the Web Ontology Language [2] seem promising directions. However, as is the case for Grid middleware, the goal is to hide the underlying complexity as much as possible from the user of a scientific workflow system.

The paper is organized as follows. In Section 2 we introduce scientific workflows by means of several real-world examples from different domains. We use those examples to illustrate some of the characteristic features and requirements of scientific workflows, and compare the latter with business workflows. In Section 3 we present specific features of Kepler and its underlying Ptolemy II system. As it turns out, Ptolemy II provides much more than a user-friendly graphical user interface (called Vergil) and a ready-to-be-extended open source platform. The main advantage of the system lies in a modeling and design paradigm called *actor-oriented modeling* that has proven to be essential to deal with the complex design issues of scientific workflows. Section 4 presents some ongoing research issues. Finally, in Section 5, we briefly summarize our findings and work.

2. SCIENTIFIC WORKFLOWS

There is a growing interest in scientific workflows as can be seen from a number of recent events, e.g. the Scientific Data Management Workshop [3], the e-Science Workflow Services Workshop [4], the e-Science Grid Environments Workshop [5], the Virtual Observatory Service Composition Workshop [6], the e-Science LINK-Up Workshop on Workflow Interoperability and Semantic Extensions [7], and last but not least, various activities as part of the Global Grid Forum (e.g. [8]), just to name a few. Scientific workflows also play an important role in a number of ongoing large research projects dealing with scientific data management, including those funded by NSF/ITR (GriPhyN, GEON, LEAD, SCEC, SEEK, ...), NIH (BIRN), DOE (SciDAC/SDM, GTL), and similar efforts funded by the U.K. e-Science initiative (myGrid, DiscoveryNet, and others). For example, the SEEK project [9] is developing an Analysis and Modeling System (AMS) that allows ecologists to design and execute scientific workflows [10]. The AMS workflow component employs a Semantic Mediation System (SMS) to facilitate workflow design and data discovery via semantic typing [12]. Thus SEEK is a good example of a community-driven project in need of a system that allows users to seamlessly access data sources and services, and put them together into reusable workflows. Indeed SEEK is one of the main projects contributing to the cross-project Kepler initiative and workflow system discussed below.

Aspects and types of workflows

Scientific workflows often exhibit particular 'traits', e.g. they can be data-intensive, compute-intensive, analysis-intensive, visualization-intensive, etc. The workflows in Sections 2.1.1–2.1.3, for example,

exhibit different features, i.e. service orientation and data analysis, re-engineering and user interaction, and high-performance computing, respectively. Depending on the intended user group, one might want to hide or emphasize particular aspects and technical capabilities of scientific workflows. For example, a 'Grid engineer' might be interested in low-level workflow aspects such as data movement and remote job control. Having workflow components (or *actors*) that operate at this level will be beneficial to the Grid engineer. Conversely, a scientific workflow system should hide such aspects from analytical scientists (say an ecologist studying species richness and productivity).

The Kepler system aims at supporting very different kinds of workflows, ranging from low-level 'plumbing' workflows of interest to Grid engineers, to analytical knowledge discovery workflows for scientists, and conceptual-level design workflows that might become executable only as a result of subsequent refinement steps [13].

In the following we first introduce scientific workflows by means of several examples taken from different projects and implemented using the Ptolemy II-based Kepler system [14]. We then discuss typical features of scientific workflows and from this derive general requirements and desiderata for scientific workflow systems. We take a closer look at underlying technical issues and challenges in Section 3.

2.1. Example workflows

2.1.1. Promoter identification

Figure 1 shows a high-level, conceptual view of a typical scientific *knowledge discovery workflow* that links genomic biology techniques such as microarrays with bioinformatics tools such as BLAST to identify and characterize eukaryotic promoters[§]—we call this the *Promoter Identification Workflow* or PIW (see also [11,15,16]: starting from microarray data, cluster analysis algorithms are used to identify genes that share similar patterns of gene expression profiles that are then predicted to be co-regulated as part of an interactive biochemical pathway. Given the gene-ids, gene sequences are retrieved from a remote database (e.g. GenBank) and fed to a tool (e.g. BLAST) that finds similar sequences. In subsequent steps, transcription factor binding sites and promoters are identified to create a promoter model that can be iteratively refined.

Although Figure 1 leaves many details open, some features of scientific workflows can already be identified: there are a number of existing databases (such as GenBank) and computational tools (such as Clusfavor and BLAST) that need to be combined in certain ways to create the desired workflow. In the past, accessing remote resources often meant implementing a *wrapper* that mimics a human entering the input of interest, submitting a HTML form, and 'screen-scraping' the result from the returned page [17]. Today, more and more tools and databases become accessible via Web services, greatly simplifying this task. Another trend are Web portals such as NCBI [18] that integrate many tools and databases and sometimes provide the scientist with a 'workbench' environment.

Figure 2 depicts snapshots of an early implementation of PIW in Kepler. Kepler is an extension of the Ptolemy II system [19] for scientific workflows. The topmost window includes a loop whose body

[§] A promoter is a consequence of a chromosome that sits close to a gene and regulates its activity.

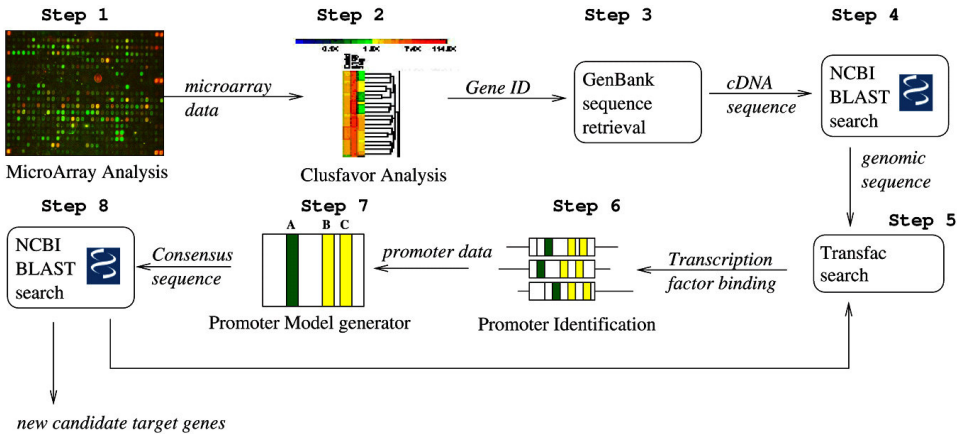


Figure 1. Conceptual ('napkin drawing') view of the PIW [11].

is expanded below and which performs several steps on each of the given gene-ids: first, an NCBI Web service is used to access GenBank data. Subsequently a BLAST step is performed to identify similar sequences to the one retrieved from GenBank. Then a second inner loop is executed (bottom window) for a transcription factor binding site analysis. Using Ptolemy II terminology, we call the individual steps *actors*, since they act as independent components which communicate with each other only through the channels indicated in the figure. The overall execution of the workflow is orchestrated by a *director* (see Section 3.3 for details).

This early PIW implementation in Kepler [11] illustrates a number of features: actual 'wiring' of a scientific workflow can be much more complicated than the conceptual view (Figure 1) suggests. A mechanism for collapsing details of a subworkflow into an abstract component (called a *composite actor* in Ptolemy II) is essential to tame complexity: the windows in Figure 2 have well-defined input and output ports and thus correspond to (sub)-workflows that can be collapsed into a more abstract, composite actor as indicated. Nevertheless, the resulting workflow is fairly complex and we will need to introduce additional mechanisms to simplify the design particularly of loops (see Section 4.1).

2.1.2. Mineral classification

The second example, from a geoinformatics domain, illustrates the use of a scientific workflow system for *automation* of an otherwise manual procedure, or alternatively, for *reengineering* an existing custom tool in a more generic and extensible environment. The upper left window in Figure 3 shows the top-level workflow: some samples are selected from a database holding experimentally determined mineral compositions of igneous rocks. These data, together with a set of classification diagrams, are fed into a Classifier subworkflow (bottom left). The manual process of classifying samples involves determining the position of the sample values in a series of diagrams such as that shown on the right in Figure 3: if the location of a sample point in a non-terminal diagram of order n has been determined

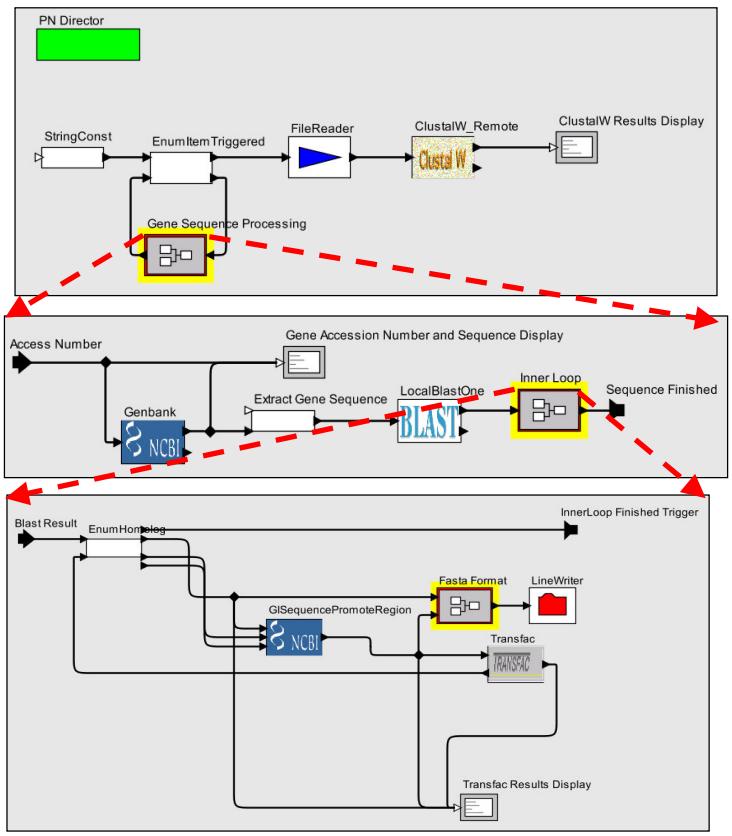


Figure 2. PIW implemented in Kepler [11]. Composite actors (subworkflows) expanded below.

(e.g. `diorite gabbro anorthosite`, bottom right), the corresponding diagram of order $n + 1$ is consulted and the point located therein. This process is iterated until the terminal level of diagrams is reached (here shown in the upper right: the classification result is `anorthosite`).

This traditionally manual process has been automated in commercial custom tools, or here in the Kepler workflow shown in Figure 3. As above, workflows are shown in graphical form using Ptolemy II's Vergil user interface [20]. Note that in Vergil, workflows can be annotated with user comments. Subworkflows (e.g. bottom left) become visible by right-clicking on a composite actor (such as Classifier, upper-left) and selecting 'Look Inside' from the resulting popup menu. Vergil also features simple VCR-like control buttons to *play*, *pause*, *resume*, and *stop* workflow execution (icons in the top-left toolbar; e.g. right-triangle for *play*).

Kepler specific features of this workflow include a searchable library of actors and data sources (*Actor* and *Data* tabs close to the upper-left) with numerous reusable Kepler actors. For example, the

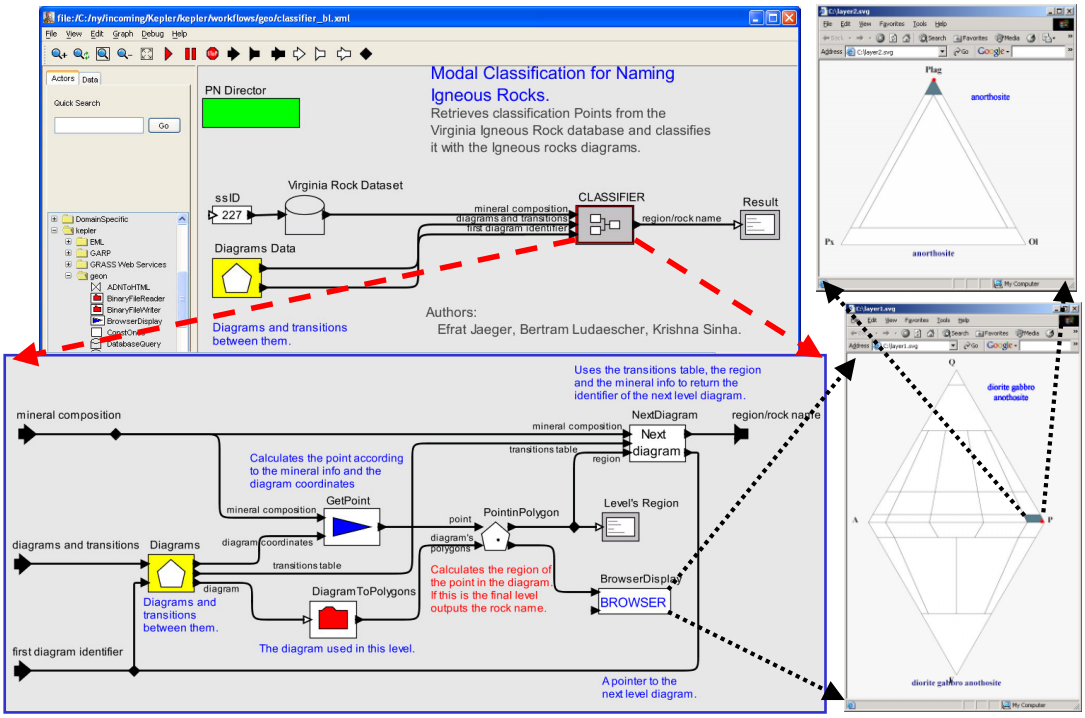


Figure 3. Mineral classification workflow (left) and generated interactive result displays (right).

Browser actor (used in the bottom right of the Classifier subworkflow) launches the user's default browser and can be used as a powerful generic input/output device in any workflow. In this example, the classification diagrams are generated on the client side as interactive SVG displays in the browser (windows on the right in Figure 3). Moving the mouse over the diagram highlights the specific region and displays the rock name classification(s) for that particular region. The Browser actor has proven to be very useful in many other workflows as well, e.g. as a device to display results of a previous step, and as a selection tool that passes user choices (made via HTML forms, check-boxes, etc.) to subsequent workflow steps.

2.1.3. *Job scheduling*

The final example workflow, depicted in Figure 4, is from a cheminformatics domain and involves running thousands of jobs of the GAMESS quantum chemical code [21] under the control of the Nimrod/G Grid distribution tool [22]. This is an example of a workflow employing *high-performance computing* (HPC) resources in a coordinated manner to achieve a computationally hard task, in this

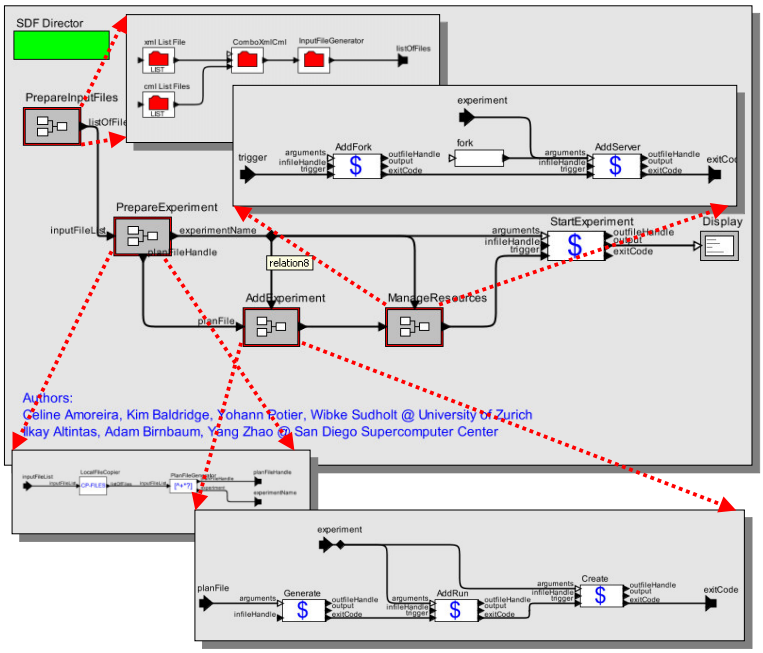


Figure 4. Workflow for scheduling HPC jobs.

case a variant of a hybrid quantum mechanics/molecular mechanics (QM/MM) technique; see [23,24] for details. Interestingly, the workflow in Figure 4 is rather domain neutral and illustrates some features typical of many high-performance computational experiments.

The main window shows four composite actors, corresponding to the four depicted subworkflows. The first one, PrepareInputs creates a list of input files for the subsequent jobs. These files are then used to create a plan file for Nimrod/G in the PrepareExperiment step. The AddExperiment subworkflow takes a plan file and generates experiment run files using several CommandLine actors. The latter is shown with a '$' icon (to indicate a command shell), and has proven to be a very useful rapid-prototyping tool: existing local applications can be made part of a workflow simply by providing a suitable command line expression and the corresponding command line arguments. The ManageResources subworkflow can create new processes (via AddFork) to run jobs and subsequently add experiments as new server processes.

This example workflow also highlights the possibility of incremental design and development: at the time of writing, not all components of the overall workflow were operational. Nevertheless, due to the clearly defined input/output interfaces of all subworkflows (a feature inherited from Ptolemy II), each of them can be designed, implemented, and tested separately. Moreover, the current version of the workflow relies heavily on invoking external applications via the CommandLine actor. Some of these applications might be 'promoted' to custom actors with native Java implementations in the future.

Such changes are encapsulated by the containing subworkflow and thus do not require changes of other parts of the workflow.

2.2. Requirements and desiderata

In this section we summarize a number of common requirements and desiderata of scientific workflows, as exhibited by the examples above or by other workflows we encountered in various application-oriented research projects including GEON, SEEK and several others [9,25–28].

R1: Seamless access to resources and services. This is a very common requirement (e.g. see the example workflows in Section 2.1), and Web services provide a first, simple mechanism for remote service execution and remote database access[¶] via service calls. However, as mentioned before, Web services are a simple solution to a simple problem. Harder problems, e.g. Web service orchestration and third-party transfer, are not solved by 'vanilla' Web services alone.

R2: Service composition and reuse, and workflow design. Since Web services emerge as the basic building blocks for distributed Grid applications and workflows, the problem of *service composition*, i.e. how to compose simple services to perform complex tasks, has become a hot research topic [34]. Among the different approaches are those that view service composition as an AI planning problem [35], a query planning problem [36,37], or a general design and programming problem. A related issue is how to design components so that they are easily reusable and not geared to only the specific applications that may have driven their original development. As we will see, service composition and reuse are addressed by employing an *actor-oriented* approach at the design level (Section 3.3), but also require flexible means for *data transformations* at the 'plumbing' level (Section 3.2).

R3: Scalability. Some workflows involve large volumes of data and/or require high-end computational resources, e.g. running a large number of parallel jobs on a cluster computer (such as workflow in Section 2.1.3). To support such data-intensive and compute-intensive workflows, suitable interfaces to Grid middleware components (sometimes called *Compute-Grid* and *DataGrid*, respectively) are necessary.

R4: Detached execution. Long running workflows require an execution mode that allows the workflow control engine to run in the background on a remote server, without necessarily staying connected to a user's client application that has started and is controlling workflow execution (such as the Vergil GUI of Kepler).

R5: Reliability and fault tolerance. Some computational environments are less reliable than others. For example, a workflow that incorporates a new Web service can easily 'break', as the latter can often fail, change its interface, or just become unacceptably slow (as it becomes more popular). To make a workflow more resilient in an inherently unreliable environment, contingency actions must be specifiable, e.g. fail-over strategies with alternate Web services.

[¶]We do not elaborate on the important challenges of *data integration* [29]; see, e.g., [30] for a survey of query rewriting techniques, and [31–33] for related issues of query capabilities and semantics, respectively.

R6: User-interaction. Many scientific workflows require user decisions and interactions at various steps[‖]. For example, an improved version of PIW (Section 2.1.1) allows the user to inspect intermediate results and select and re-rank them before feeding them to subsequent steps. An interesting challenge is the need for user interaction in a detached execution. Using a notification mechanism, the user might be asked to reconnect to the running instance and make a decision before the paused (sub-)workflow can resume.

R7: 'Smart' re-runs. A special kind of user interaction is the change of a parameter of a workflow or actor. For example, in a visualization pipeline or a long-running workflow, the user might decide to change some parameters after inspecting intermediate or even final results. A 'smart' re-run would not execute the workflow from scratch, but only those parts that are affected by the parameter change. In dataflow-oriented systems (e.g. visualization pipeline systems such as AVS, OpenDX, SCIRun, or the Kepler system) this is easier to realize than in more control-oriented systems (e.g. business workflow systems), since data and actor dependencies are already explicit in the system. Another useful technique in this context is *checkpointing*, which allows one to backtrack (in the case of a parameter change or even a system failure; cf. (R5)) to a previously saved state without starting over from scratch.

R8: 'Smart' (semantic) links. A scientific workflow system should assist workflow design and data binding phases by suggesting which actor components might possibly fit together (this is also an aspect of (R2), service composition), or by indicating which data sets might be fed to which actors or workflows. To do so, some of the semantics of data and actors has to be captured. However, capturing data semantics is a hard problem in many scientific disciplines: e.g. measurement contexts, experimental protocols, and assumptions made are often not adequately represented. Even if corresponding metadata are available, it is often not clear how to best make them useable by the system. It seems clear though that ontologies provide a very useful *semantic type system* for scientific workflows, in addition to the current (structural) type systems [12].

R9: Data provenance. Just as the results of a conventional wet lab experiment should be reproducible, computational experiments and runs of scientific workflows should be reproducible and indicate which specific data products and tools have been used to create a derived data product. Beyond the conventional capture of metadata, a scientific workflow system should be able to automatically log the sequence of applied steps, parameter settings and (persistent identifiers of) intermediate data products. A related desiderata is *automatic report generation*: the system should allow the user to generate reports with all relevant provenance and runtime information, e.g. in XML format for archival and exchange purposes and in HTML (generated from the former, e.g. via an XSLT script) for human consumption.
Data provenance can be seen as a prerequisite to (R8): in order to provide semantic information about a derived data product, suitable provenance information is needed.

While the above list of requirements and desiderata for scientific workflow systems is by no means complete, it should be sufficient to capture many of the core characteristics. Other requirements include

[‖]In fact, when workflow management was still called 'office automation', humans were the main processors of tasks—the workflow system was just used for book-keeping; cf. Section 2.3.

the use of an intuitive GUI to allow the user to compose a workflow visually from smaller components, or to 'drill-down' into subworkflows, to animate workflow execution, to inspect intermediate results, etc.

A scientific workflow system should also support the combination of different *workflow granularities*. For example, coarse-grained workflows, akin to Unix pipelines or Web service-based workflows, consist mainly of 'black box' actors whose contents are unknown to the system. Scientific workflows may also be very fine grained, or include fine-grained subworkflows. In that case, components are 'white boxes' containing, e.g., the visual programming equivalent of an algorithm, or a system of differential equations to be solved, in other words, a detailed specification known to the system.

2.3. Differences to business workflows

The characteristics and requirements of scientific workflows are partially overlapping those of *business workflows*. Indeed, the term 'scientific workflows' seems to indicate a very close relationship with the latter, while a more detailed comparison reveals a number of significant differences. Historically, business workflows have roots going back to *office automation* systems of the 1970s and 1980s, and gained momentum in the 1990s under different names, including *business process modeling* and *business process engineering*; see, e.g., [38–40].

Today we see some influence of business workflow standards in the Web services arena, specifically standards for *Web service choreography*[**]. For example, the Business Process Execution Language for Web Services (BPEL4WS) [42], a merger of two earlier standards, IBM's WSFL and Microsoft's XLANG, has received some attention recently.

When analyzing the underlying design principles and execution models of business workflow approaches, a focus on *control-flow* patterns and *events* becomes apparent, whereas dataflow is often a secondary issue. For example, [43] describe a large number of workflow design patterns that can be used to analyze and compare business workflow standards and products in terms of their control features and expressiveness.

Scientific workflow systems, on the other hand, tend to have execution models that are much more *dataflow oriented*. This is true, e.g., for academic systems including Kepler, Taverna [44], and Triana [45], and for commercial systems such as Inforsense's DiscoveryNet or Scitegic's Pipeline-Pilot. With respect to their modeling paradigm and execution models, these systems seem closer to an 'AVS for scientific data and services' than to the more control-flow and task-oriented business workflow systems, or to their early scientific workflow predecessors [46–48].

The difference between dataflow orientation and control-flow orientation can also be observed in the underlying formalisms. For example, visualizations of business workflows often resemble flowcharts, state transition diagrams, or UML activity diagrams, all of which emphasize events and control flow over dataflow. Formal analysis of workflows usually involves studying their control flow patterns [49], and is often conducted using Petri nets.

[**]Despite the long history of business workflows, it is surprising how short lived some of the so-called standards are, as '*most of them die before becoming mature*' [41].

Conversely, the underlying execution model of current scientific workflow systems usually resembles or is even directly implemented as a *dataflow process network* [50,51], having traditional application areas, e.g. in digital signal processing. Dataflow-oriented approaches are applicable at very different levels of granularity, from low-level CPU operations found in certain processor architectures, to high-level programming paradigms such as *flow-based programming* [52]. Scientific workflow systems and visualization pipeline systems can also be seen as dataflow-oriented *problem solving environments* [53] that scientists use to analyze and visualize their data. Last but not least, there is also a close relationship between dataflow-oriented approaches and (pure) functional languages, including non-strict variants such as Haskell (cf. Section 4.1).

3. HIGHLIGHTS OF KEPLER

In this section, we discuss some highlights of the current Kepler system as well as some upcoming extensions. Many features directly address the requirements and desiderata from Section 2. More research-oriented extensions are described in Section 4.

3.1. Web service extensions

A basic requirement for scientific workflows is seamless access to remote resources and services (see (R1) in Section 2.2 and the examples in Section 2.1). Since Web services are emerging as the standard means for remote service execution of loosely coupled systems, we extended Kepler early on to handle Web services. Given the URL of a Web service description [1], the generic WebService actor of Kepler can be *instantiated* to any particular operation specified in the service description. After instantiation, the WebService actor can be incorporated into a scientific workflow as if it were a local component. In particular, the WSDL-defined inputs and outputs of the service are made explicit via the instantiated actor's input and output ports.

Figure 5 shows screenshots of an extended Web service harvesting feature, implemented by a special Web service Harvester component[††]. As in the case of the generic WebService actor, a URL is first provided (see (1) in Figure 5), however this time not to an individual WSDL description of a Web service, but to a Web service repository. The repository URL might point to a UDDI repository, or simply to a Web page listing multiple WSDL URLs as shown in (2). The Harvester then retrieves and analyzes all WSDL files of the repository, creating instantiations of Web service actors in the user's local actor library; see (3). For example, one of the harvested services, the BLAST Web service, comprises five service operations which are imported into a corresponding subdirectory. The user can then drag and drop any of these service operations on the workflow canvas for use in a scientific workflow (4). The Harvester feature facilitates rapid prototyping and development of Web-service-based applications and workflows in a matter of minutes—that is, provided:

(i) the Web services are alive when needed; and
(ii) they can be wired together more or less directly to perform the desired complex task.

[††]Inspiration came from a similar feature in Taverna.

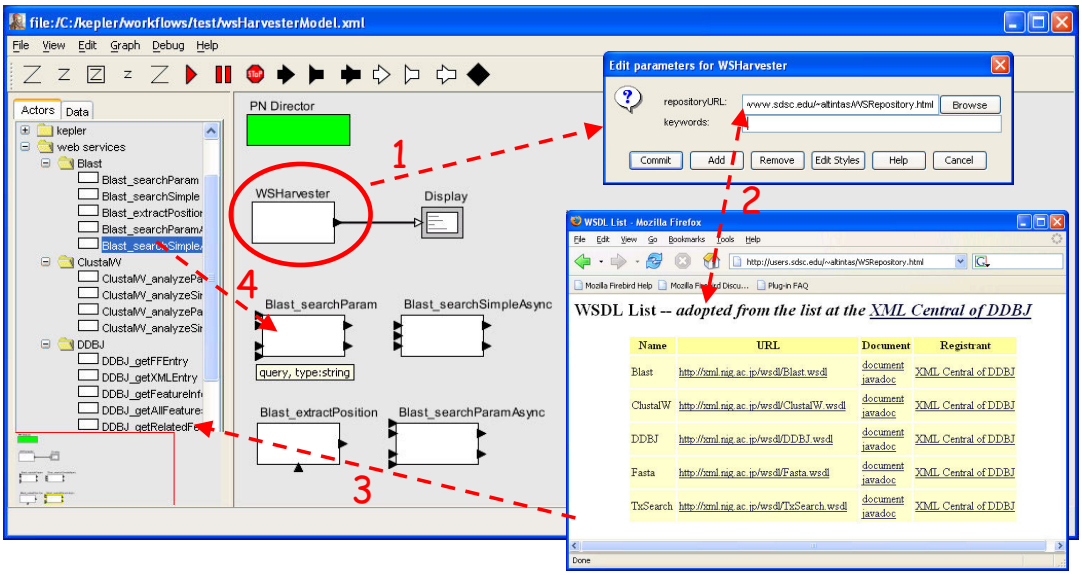


Figure 5. Kepler Web service Harvester in action: repository access (1–2), harvesting (3), and use (4).

The problem with (i) is that, while harvested Web services look like local components, their runtime failure can easily 'break' a scientific workflow, reminding the user that the service *interface* has been harvested, not the actual code[‡‡]. We are currently extending Kepler to make workflows with Web services more reliable. One simple approach is to avoid the association of a service operation with a fixed URL. Instead, a list of alternative services can be provided when the workflow is launched, and service failure can then be compensated by invocation of one of the alternative services. Another option is to insert special control tokens into the data stream, indicating to downstream actors the absence of certain results. Long-running workflows may thus more gracefully react to Web service failures and produce at least partial results. This idea has been further developed for 'collection-oriented' (in the functional programming sense) workflows: via so-called 'exception-catching actors', invalid (due to failures) data collections can be filtered out of the data stream, while valid subcollections pass through unaffected [54]. An interesting research question is how to extend Ptolemy II's pause–resume model to a fully fledged transaction model that can handle service failures.

The problem (ii) is even more fundamental and has different aspects: at the *design level* the challenge is how to devise actors that can be reused easily. In Section 3.3 we give a brief introduction to *actor-oriented modeling*, the underlying paradigm of Ptolemy II, and discuss how it facilitates component composition and reuse. At the '*plumbing*' *level* it is often necessary to apply data transformations

[‡‡]Which is of course the whole point of Web services.

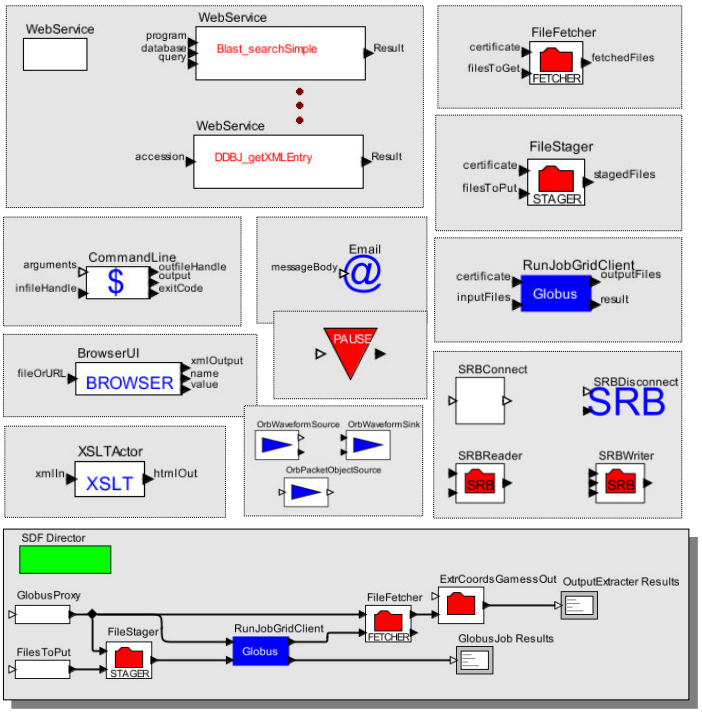


Figure 6. Grid actors and other Kepler extensions.

between two consecutive Web services (called 'shims' in Taverna). Such data transformations are supported through various actors in Kepler, e.g. XSLT and XQuery actors to apply transformations to XML data, or Perl and Python actors for text-based transformations.

3.2. Grid and other extensions

Figure 6 depicts a number of Kepler actors that facilitate scientific workflows, including workflows that make use of 'the Grid'. In the upper left, the previously discussed generic WebService actor and some instantiations are shown. Note how the latter specialize their actor interface via their input/output ports: for example, Blast_SearchSimple has three input ports and one output port, for the search arguments and result, respectively. The naming scheme used is *WSN_OP*, where *WSN* is the name of the Web service and *OP* is a specific Web service operation.

The upper right shows two Grid actors, called FileFetcher and FileStager, respectively. These actors make use of GridFTP [55] to retrieve files from, or move files to, remote locations on the Grid. The GlobusJob actor below is another Grid actor, in this case for running a Globus job [56]. At the bottom of Figure 6 a small workflow is shown that takes a Globus proxy and some input files, staging

the files to where the job is run, then fetching the results from the remote location and displaying them on the client side. The director box specifies that this workflow is executed using a SDF (Synchronous Data-Flow) director. This director analyzes the dataflow dependencies and token consumption and production rates of actors (here, token = file), and schedules the execution of actors accordingly.

On the right, a number of actors that use the SDSC Storage Resource Broker [57] are shown, e.g. to connect and disconnect from SRB and to get and put files from and to SRB space, respectively. We are currently in the process of providing all commonly used SRB commands as actors. This will allow the Kepler user to design and execute Grid workflows involving a number of different tools, e.g. SRB for data-handling aspects, and Globus, Nimrod and other tools for computational aspects and job scheduling.

In the center and left of Figure 6, various other Kepler actors are shown. The CommandLine actor can be used to incorporate any application into a workflow, provided it can be accessed from the command line*. The '`$`' icon is reminiscent of a shell prompt. The actor is parameterized with the arguments of the shell command, making it easy to create generic or specialized command line invocations. A Browser actor is shown directly below (cf. Section 2.1.2). It takes as input an HTML file or URL and displays it in the user's default browser. This makes the actor an ideal output device for displaying intermediate or final workflow results in ways that are well known to users. Another extremely useful application of this actor is as an input device for user interactions. The result file of an upstream actor might have been transformed to a HTML file (e.g. using the XSLT actor) and augmented with HTML forms, check boxes, or other input forms that are displayable to the user in a standard Web browser. Upon executing the desired user interaction, a `http-post` request is sent to a special Kepler Web server, acting as a listener, and from there the workflow is resumed.

The Email actor in the center of the figure provides a simple notification mechanism to inform the user of specific situations in the workflow. Together, the Email and Browser actors address core issues of requirement (R6) in Section 2.2. The Pause actor (down-triangle) pauses workflow execution at specific points, allowing the user to inspect intermediate results, possibly changing parameter values, and resuming the workflow subsequently (addressing (R7) in Section 2.2).

Finally, actors for accessing *real-time data streams* from ROADNet sensor networks [28] have recently been added. These actors (e.g. OrbWaveformSource) can be integrated easily into Kepler, since many of the underlying Ptolemy II directors support streaming execution†.

3.3. Actor-oriented modeling

Arguably the most unique feature of Kepler comes from the underlying Ptolemy II system:

> The focus [of the Ptolemy project] is on assembly of concurrent components. The key underlying principle ... is the use of well-defined *models of computation* that govern the interaction between components‡.

*For example, Kepler workflows can include data analysis steps via calls to R [58].
†This should come as no surprise, since dataflow process networks are defined on token streams in the first place.
‡http://ptolemy.eecs.berkeley.edu/objectives.htm.

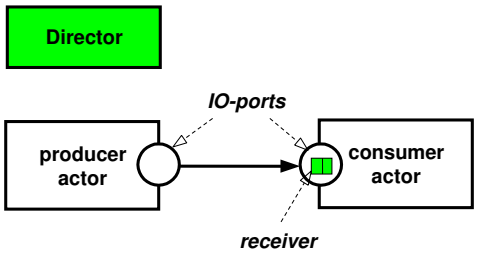


Figure 7. The semantics of component interaction is determined by a director, which controls execution and supplies the objects (called receivers) that implement communication.

This focus together with the *actor-oriented modeling* paradigm make Ptolemy II an ideal starting point for tackling the breadth of challenges in scientific workflow design and execution. In Ptolemy, a system or model thereof (in our case, a scientific workflow) is viewed as a composition of independent components called *actors*. Communication between actors happens through interfaces called *ports*. We distinguish between *input ports* and *output ports*. In addition to the ports, actors have *parameters*, which configure and customize the behavior[§]. For example, a generic filter actor might consume a stream of input tokens via an input port, letting through to the output port only those tokens that satisfy a condition specified by a parameter.

Actors, or more precisely their ports, are connected to one another via *channels*. Given an interconnection of actors, however, there are many possible execution semantics that one could assign to the diagram. For example, actors might have their own thread of control, or their execution might be triggered by the availability of new inputs.

A key property of Ptolemy II is that the execution semantics is specified in the diagram by an object called a *director* (see Figure 7). The director defines how actors are executed and how they communicate with one another. Consequently, the execution model is less an emergent side effect of the various interconnected actors and their (possibly ad hoc) orchestration, and more a prescribed concurrent semantics as one might find in a well-defined concurrent programming language. The execution model defined by the director is called the *model of computation*. Patterns of concurrent interaction are factored out into the design of the directors, rather than being individually constructed by the designer of the workflow. Figure 7 depicts a producer and a consumer actor whose ports are connected by a unidirectional channel. The diagram is annotated by a director, which might, for example, execute the producer prior to the consumer so as to respect data precedences. The communication between the actors is mediated by an object called a *receiver*, which is provided by the director, not by the actors. Thus, for example, whether the communication is buffered or synchronous is determined by the designer of the director, not by the designer of the actor. This hugely improves the reusability of actor designs.

[§]Parameters are usually not shown in the figures.

Process networks

The Process Network (PN) director is a popular choice for designers of scientific workflows. It gives a diagram of the semantics of (dataflow) process networks [50,51]. In this semantics, actors are independent processes that execute concurrently, each with its own thread of control, and communicate by sending *tokens* through unidirectional channels with (in principle) unbounded buffering capacity. Writing to a channel is a non-blocking operation, while reading from a channel can block until sufficient input data are available. This model of computation is similar to that provided by Unix pipes, as in the following example of a Unix command-line composition of processes:

```
cat foo.txt | bar | baz
```

This example shows three independently executing processes (`cat`, `bar`, and `baz`) that are connected to one another through unidirectional pipes. The stream of tokens flowing between the processes also synchronizes them if necessary. For example if `bar` and `baz` are filter operations working on a single line of text at a time (e.g. `grep` *xyz*), then a Unix process executing `bar` will block until a line of text is provided by the process executing `cat foo.txt`. Unlike Unix pipes, however, the PN director in Ptolemy II tolerates feedback loops and forking and merging of data streams. It performs deadlock detection, and manages buffers to keep memory requirements bounded (if possible).

The PN director is only one example of a large number of directors available in Ptolemy II. There is also, for example, the Synchronous Data-Flow (SDF) director, which can be used for specialized process networks with fixed token production and consumption rates per firing (see below). The SDF director performs static analysis on a workflow that guarantees absence of deadlocks, determines required buffer sizes, and optimizes the scheduling of actor execution. Other directors have been constructed for modeling Discrete Event (DE) systems, Continuous-Time (CT) models (which solve ordinary differential equations), and Communication Sequential Processes (CSPs), to mention just a few [20].

By relieving actors from the details of component interaction, the actors themselves become much more reusable (cf. (R2) in Section 2.2). The behavior of an actor adapts to the execution and communication semantics provided by the director. This feature of actor-oriented modeling is called *behavioral polymorphism*. For example, a single Ptolemy II actor implementation of an arithmetic operation, say Plus, can be connected to any number of input operands and reused within different models of computation and under the control of different directors. An SDF director, for example, schedules the actor invocation (or 'firing') as soon as *all* inputs have data, which it knows since actors declare their fixed token consumption and production rates in the SDF domain. In contrast, when the Plus actor is governed by a DE director, additions happen when *any* input has data, corresponding to the different overall execution model in the DE domain. In addition to behavioral polymorphism, the Ptolemy II type system also supports *data polymorphism*, again increasing the reusability of actors. For example, our Plus actor can be implemented in such a way that it dynamically chooses the correct numeric addition (integer, float, double, complex), depending on the types of inputs it receives. Moreover, on other data types, e.g. strings, vectors, matrices, or user-defined types, the Plus actor[¶] can execute appropriate actions, e.g. string concatenation, vector or matrix addition, etc.

[¶]This actor is called AddSubtract in Ptolemy II.

execution ⟶ **preinitialize**, *type-check*, *run**, **wrapup**
run ⟶ **initialize**, *iteration**
iteration ⟶ **prefire**, **fire***, **postfire**

Figure 8. AOPI execution *phases* and actor **methods**.

Actor-oriented programming interface

Actor-oriented modeling addresses several challenges in the design of complex systems [59]. We have already mentioned improved component reusability due to behavioral and data polymorphism. Another aspect is *hierarchical modeling*. As illustrated by the examples in Section 2.1, subworkflows can be abstracted into (composite) actors themselves (e.g. see the Classifier actor/subworkflow in Figure 3) and thus arbitrarily nested. In the following, we give a simplified introduction on some implementation aspects of Ptolemy II's actor-oriented approach. These can be adapted to the context of scientific workflows and distributed, service-oriented environments, leading to a more structured approach to service composition and workflow design.

The structure we propose is based on various phases and methods in Ptolemy II's actor-oriented programming interface (AOPI); see Figure 8. These AOPI methods are used by a director to orchestrate overall execution. Symbols in boldface denote actual methods that actor implementations have to provide; the remaining symbols describe other phases[‖] of the overall execution.

When a director starts a workflow *execution*, it invokes the **preinitialize** method of all actors. Since this method is invoked only once per lifetime of an execution (even if there are multiple runs), and prior to all other activities, this is a good time to put in place the receiver components of actors, and for actors to 'advertise' their supported port data types, transport protocols, etc.

Next the director *type-check*s all connections and ports. This includes checking each port's data types, all (previously advertised) type constraints, and the validity of port types being connected through channels. A type inference algorithm is used to determine the most general types satisfying the given constraints. For scientific workflows, we can modify directors to also type-check which transport protocol to use, or to check whether producer and consumer actors exchange data directly or via *handles*[**]. For example, if an actor A declares its output port to be of *handle type* '`http | ftp`' and a connected actor B declares its input port to be of handle type '`http`', then type-checking can establish that the connection is valid, provided A's output port is subtyped to use `http` handles only. Indeed such information can and should be passed to the actor with the invocation of the **initialize** method.

Other possible actions during execution of **initialize** include the following. Web service actors can 'ping' the Web services they represent and signal failure-to-initialize if the corresponding service is not alive. A 'fail-over-aware' director can use this information to replace the defective Web service with an equivalent one that is alive (see (R5) in Section 2.2). A workflow *execution* will often consist

[‖]Some correspond to methods of other Ptolemy II entities, e.g. *director methods* or *manager methods* [20].
[**]By *handle* we mean a unique identifier that can also be used to retrieve data, e.g. a URL.

only of one *run*, but if a workflow is re-run, **initialize** is called again. A *run* usually includes multiple *iteration*s, each of which includes a call to **prefire**, **fire** (possibly called repeatedly by some special directors), and a call to **postfire**. The main actor operation finally happens in the **fire** method, e.g. a Web service actor will make the actual remote service call here.

Towards actor-oriented scientific workflows

The idea of *actor-oriented scientific workflows* is to apply the principles of actor orientation and hierarchical modeling, underlying the Ptolemy approach [20,59], to the modeling and design of scientific workflows. In particular, Web service operations, which provide the building blocks of many loosely coupled workflows, should be structured into different parts, corresponding to the different phases and methods used in actor-oriented modeling. For example, to implement a Web service w_A, the service developer should think of specific Web service operations such as w_A.**initialize** and w_A.**prefire** in addition to the main 'worker' method w_A.**fire**. As in the case of Ptolemy actors, this will lead to more generic and reusable components and even facilitate more complex extensions such as stateful Web services[††].

4. RESEARCH ISSUES

In this section we briefly discuss some technical issues that we have begun addressing for Kepler, but that are less mature and require some additional research.

4.1. Higher-order constructs

The early implementation of the PIW depicted in Figure 2 demonstrated the feasibility and some advantages of implementing scientific workflows in the Kepler extension of Ptolemy II [11]. However, it also highlighted some inherent challenges of the dataflow-oriented programming paradigm [60]. We have argued in Section 2.3 that many current scientific workflow systems are more dataflow oriented than business workflow systems and approaches, which tend to emphasize event-based control-flow rather than dataflow. When designing real-world scientific workflows it is necessary, however, to handle complex control-flows within a dataflow-oriented setting as well. It is well known that control-flow constructs require some thought in order to handle them properly. The fairly intricate network topology in Figure 2 includes backward-directed 'dataflow' channels, having the sole purpose of sending control tokens that initiate another iteration of a subworkflow. While such complicated structures achieve the desired effect (here, a special kind of loop), they are hard to understand, design, and maintain. Such ad hoc constructions also increase the complexity of workflow design while diminishing the overall reusability of workflow components (see (R2) in Section 2.2). Fortunately, there are better ways to incorporate *structured control* into a dataflow-oriented system, thereby directly supporting workflow design as required by (R2).

[††]Statefulness is an established concept in actor-oriented modeling and dataflow networks; e.g. it can be represented explicitly via feedback loops.

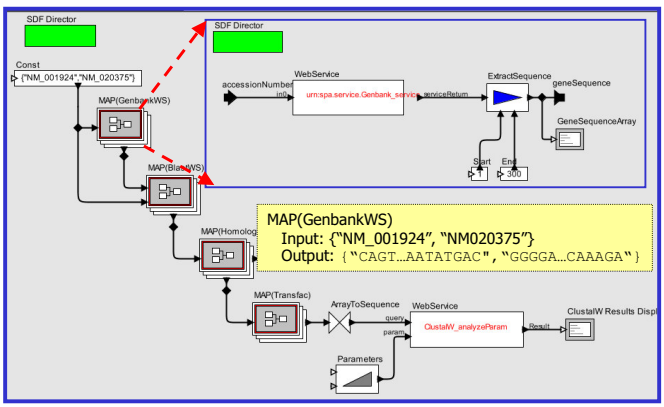


Figure 9. PIW variant with map iterator.

In [60] we have illustrated how higher-order functional programming constructs can be used to improve the design of PIW. In particular, the higher-order function map :: $(\alpha \to \beta) \to [\alpha] \to [\beta]$ has proven to be very useful to implement a certain type of iteration. It takes a function f (from α to β) and a list of elements of type α, and applies f to each list element, returning the list of result elements (each of type β). Thus map is defined as

$$\text{map } f\ [x_1, x_2, \ldots, x_n] = [f(x_1), f(x_2), \ldots, f(x_n)]$$

For example, map $f\ [1, 2, 3] = [1, 4, 9]$ for $f(x) = x^2$.

Figure 9 shows an improved version of the PIW workflow from Section 2.1.1 and Figure 2, now using the higher-order map function. Note how backward-directed flows of control tokens are avoided. Instead, iterations are realized as nested subworkflows inside a higher-order Map actor. For example, to implement a look-up of a list of gene sequences via a GenBank Web service that can only accept one gene at a time, we simply create the higher-order construct MAP(GenBankWS) as shown in Figure 9 (the 'stack' icon indicates that the contained workflow is applied multiple times).

Other higher-order functional programming constructs, e.g. foldr (for 'fold right') can be similarly used to provide more abstract and modular iteration and control constructs in a dataflow setting, and we plan to add those to Kepler in the future. The utility of declarative functional programming methods for dataflow-oriented systems is no coincidence; see, e.g., [61] for more on the close links between dataflow, functional, and visual programming, and [62] for interesting applications in implicit parallel programming. Here we only give a simple illustration using a core subworkflow of PIW in a Haskell specification; see [60] for details:

```
d0 = $Gid                          % input: some gene-id
d1 = genBankG in                   % get its gene sequence
d2 = blastP d1          % find candidates from similar seqs
d3 = map genBankP d2       % get promoter sequences
d4 = map promoterRegion d3          % compute regions
d5 = map transfac d4     % compute transcr. factor sites
d6 = zip d2 d4 % create list of (promoter-id,region) pairs
d7 = map gpr2str d6          % accumulate into string list
d8 = concat d7                        % create a single file
d9 = putStr d8               % output to subsequent steps
```

The input and output (ports) of this workflow are given by d0 and d9, respectively. Note the use of map to iterate over lists where the available services (e.g. genBankP) can only handle one item at a time. Also note that these 10 equations establish a simple forward-only dataflow process network with the d_i representing named channels, and the expressions on the right of the equation representing processes (i.e. actors). A merge of two parallel branches happens, e.g. through the function zip that creates a single stream of pairs (*promoter-id*, *promoter-region*) in channel d6 from the two streams in d2 and d4.

4.2. Third-party transfers

Scientific workflows can involve large volumes of data (see (R3) in Section 2.2). In a Web service setting, this creates a problem since so-called third-party transfers are not currently supported by Web services: let us consider two Web services w_A and w_B, located at two sites s_1 and s_2, respectively. w A takes some input x and produces some data d that we would like to pass on to w_B, which produces the final output data y. We can depict this as follows:

$$\xrightarrow{x} \boxed{w_A@s_1} \xrightarrow{d} \boxed{w_B@s_2} \xrightarrow{y}$$

Assume that the overall execution of this workflow WF is coordinated and controlled by a workflow engine E (e.g. Kepler) running at some site s_3. Current Web service implementations do not allow the engine E to call $w_A@s_1$, telling it to route d directly to $w_B@s_2$. Instead, Web service invocations, and the input/output dataflows that go with them, all go through $E@s_3$. In pseudo-code this means:

$$\begin{aligned} WF@s_3(\textbf{in } x, \textbf{out } y) = \{ \\ d@s_3 := w_A@s_1(x@s_3); \\ y@s_3 := w_B@s_2(d@s_3) \ \} \end{aligned}$$

How do we execute the 'remote assignments' shown here? To execute $WF@s_3$, the workflow engine E first sends a request message containing x to $w_A@s_1$. Upon completion, w_A replies back to $E@s_3$ with the result d. Now $WF@s_3$ can proceed and E forwards d to s_2 where w_B can work on it. The final result y is then sent from s_2 back to s_3. This simple call/return execution is quite desirable from a modeling and design point of view since control-flow and dataflow go hand in hand, and since the control engine E does not have to worry about the status of direct (i.e. third party) transfers of data d from w_A to w_B. The downside, however, is that data are moved around more often than necessary. Let us trace the 'data shipments' of x, d, and y:

1. **ship** $x@\mathsf{s}_3 \rightsquigarrow x@\mathsf{s}_1$ *% part of request to* w_A
2. $@\mathsf{s}_1$ **execute** $d := \mathsf{w}_A(x)$ *% execute* w_A
3. **ship** $d@\mathsf{s}_1 \rightsquigarrow d@\mathsf{s}_3$ *% part of reply from* w_A
4. **ship** $d@\mathsf{s}_3 \rightsquigarrow d@\mathsf{s}_2$ *% part of request to* w_B
5. $@\mathsf{s}_2$ **execute** $y := \mathsf{w}_B(d)$ *% execute* w_B
6. **ship** $y@\mathsf{s}_2 \rightsquigarrow y@\mathsf{s}_1$ *% part of reply from* w_B

If d is very large, executing both steps (3) and (4) is wasteful: first d is sent from s_1 to s_3 where the workflow engine E runs, only to be sent to s_2 in the next step. Instead of sending d over the wire twice, the more direct third-party transfer

$$\mathsf{w}_A@\mathsf{s}_1 \overset{d}{\rightsquigarrow} \mathsf{w}_B@\mathsf{s}_2$$

moves d only once, but as mentioned before, is not currently supported by Web services[‡‡]. The question becomes: how can we avoid unnecessary transfers and achieve the efficiency of third-party transfer, while retaining the above simple call/return execution model?

A handle-oriented approach

A simple solution to the above problem is that w_A does not send the actual data d but a *handle* h_d to them. Such a handle corresponds to a 'logic pointer' and can be represented by a globally unique URI, but may also be a URL and indicate the protocol by which d is to be accessed, e.g. `http`, `ftp`, `GridFTP00`, `scp`, or `SRB`. If we replace all data occurrences x, d, and y by handles h_x, h_d, and h_y, respectively, we obtain the following:

1. **ship** $\mathsf{h}_x@\mathsf{s}_3 \rightsquigarrow \mathsf{h}_x@\mathsf{s}_1$ *% request to* w_A
2. $@\mathsf{s}_1$ **execute** $\mathsf{h}_d := \mathsf{w}_A(\mathsf{h}_x)$ *% execute* w_A
3. **ship** $\mathsf{h}_d@\mathsf{s}_1 \rightsquigarrow \mathsf{h}_d@\mathsf{s}_3$ *% reply from* w_A
4. **ship** $\mathsf{h}_d@\mathsf{s}_3 \rightsquigarrow \mathsf{h}_d@\mathsf{s}_2$ *% request to* w_B
5. $@\mathsf{s}_2$ **execute** $\mathsf{h}_y := \mathsf{w}_B(\mathsf{h}_d)$ *% execute* w_B
6. **ship** $\mathsf{h}_y@\mathsf{s}_2 \rightsquigarrow \mathsf{h}_y@\mathsf{s}_1$ *% reply from* w_B

Now, instead of sending (the possibly very large) d over the wire twice in (3) and (4), we only do so for the (constant size) handle h_d. We cannot hope to further reduce this since a reply message from w_A to E and a new request from E to w_B are necessary for the overall control of workflow execution.

In order to implement the above handle solution, we need to slightly extend our Web services: in steps (2) and (5), w_A and w_B need to process handles by dereferencing them or by creating new ones. The former happens when a Web service acts as a *consumer* of data (w_A consumes x), while the latter is needed in the role of a data *producer* (w_A produces d).

Consider, for example, the case where handles are represented as URLs with `http` as the transport protocol. In step (2) above, w_A needs to dereference h_x before it can execute its function. h_x might be, for instance, http://foobar.com/f17. When dereferenced via http-get it yields the actual data x^*.

[‡‡]And even if it were, 'divorces' control-flow and dataflow, resulting in more complex execution models.
[*]Note that while the handle h_x is sent from s_3 to s_1 in step (1), x might actually *not* reside at s_3.

To properly process handles as a data consumer, the operation 'receive x' has to be replaced by 'receive h_x', followed by a 'dereference and get' operation x := `http-get`(h_x). All subsequent read operations can then operate on x as before.

In the role of a data producer, we have the reverse situation. We want to avoid shipping of the actual result data d and instead send a handle h_d. Thus, we need to first create this handle, e.g. by creating a new file f18 that can be accessed via h_d = http://baz.edu/f18. All subsequent write access to d will proceed unchanged, provided the file name `f18` is used for d. Finally, we need to replace 'send d' with 'send h_d'.

We are currently working on extensions of Kepler that make the system 'handle aware' [63]. For example, during the type-checking phase (Figure 8) a handle-aware director could determine whether two Web service actors A and B that invoke the Web services w_A and w_B, respectively, support compatible handle types. For this to work seamlessly, Web services themselves should offer an actor-oriented programming interface as presented in Section 3.3.

4.3. Other research issues

Higher-order constructs and the handle approach to third-party transfers are only two of a number of pressing research issues in scientific workflows[†]. For example, detached execution (R4), reliability and fault tolerance (R5), semantic links (R8), and data provenance (R9) are all scientific workflow requirements that need further attention in the future. For example, [12] presents some initial work on the use of ontologies as semantic types to help generate data transformation mappings between consecutive workflow steps. These kinds of semantic extensions can help at both levels, at the 'plumbing' level to create data transformations as in [12], and at the design level to create more reusable components (R2) and to support 'smart' links in workflows (R8).

4.4. Related work

In Section 3 we have described some of the features of Kepler and the underlying Ptolemy II system on which Kepler is based. Ptolemy II aims at modeling and design of heterogeneous, concurrent systems. In contrast, Kepler aims at the design and execution of scientific workflows. Consequently, Kepler extensions to Ptolemy II include numerous actors and capabilities that facilitate scientific workflows (e.g. Web service actors and harvester, GridFTP, SRB and database actors, command-line and secure shell actors, etc.) Additional components are constantly added, e.g. to support statistics packages (such as R), GIS functionality (e.g. Grass and ArcIMS couplings), and other scientific data analysis and visualization capabilities [64].

The research and development on Kepler also benefits from interactions and collaborations with other groups. On the one hand, development is driven by application scientists, the ultimate 'customers' of scientific workflow systems, on the other hand, work in related projects also influences Kepler developments. For example, Taverna [44,65] is a system that focuses on Web service-based bioinformatics workflows. In contrast, Triana [45,66] provides mechanisms for coupling workflows

[†]Addressing (R2) and (R3), respectively.

more tightly with Grid middleware tools. Cross-fertilization between these and other projects has happened, e.g. through e-Science LINK-UP workshops [7], meetings and workshops at GGF [8], etc. Other scientific workflow tools include Pegasus [67], Chimera, and job scheduling tools such as Condor/G [68] and Nimrod/G [22]. For a taxonomy of workflow management systems for Grid computing and a comparison of systems, see [69]. Future work will address the various outstanding research issues and workflows requirements that have not yet been (fully) met. For example, some projects contributing to Kepler plan to provide couplings to highly-interactive visualization tools such as SCIRun [64] and GeoVista [70].

5. CONCLUSIONS

We have provided an overview of scientific workflow management issues, motivated by real-world examples that we encountered in a number of application-oriented projects. The spectrum of what can be called a scientific workflow is wide and includes scientific discovery workflows (e.g. Section 2.1.1), workflows that automate manual procedures or reengineer custom tools (e.g. Section 2.1.2), and data and compute-intensive workflows (e.g. Section 2.1.3). Scientific workflow support is needed for practically all information-oriented scientific disciplines, including bioinformatics, cheminformatics, ecoinformatics, geoinformatics, physics, etc. We identified a number of common requirements and desiderata of scientific workflows (Section 2.2) and contrasted them with business workflows.

The Kepler system addresses many of the core requirements (Section 3) and provides support for Web service-based workflows and Grid extensions. The source code of Kepler is freely available [14] and a first alpha-release was distributed earlier this year. A unique feature of Kepler is inherited from the underlying Ptolemy II system: the actor-oriented modeling approach. This approach facilitates modeling and design of complex systems and thus also provides a very promising direction for pressing problems such as Web service composition and orchestration. The way data polymorphism and behavioral polymorphism are supported by an actor-oriented approach, which 'concentrates' component interaction in a separate *director* entity, can also shed light on other efforts to create reusable component architectures such as CCA [71]. Areas of research include modeling issues such as the use of higher-order functional constructs for workflow design (Section 4.1), and optimization issues such as the use of virtual data references (handles) to facilitate data-intensive, Web service-based workflows (Section 4.2).

ACKNOWLEDGEMENTS

Kepler is an open source, cross-project collaboration that would not exist without the contributions of the many team members. We thank all current and past contributors to Ptolemy II—the Kepler systems would not be possible without them. We also thank all Kepler members for their contributions, in particular, Tobin Fricke for implementing actors that access the wonderful world of ROADNet real-time data streams, Steve Neuendorffer and Christopher Brooks for sharing their insights into Ptolemy II, Rod Spears for QBE facilities, Xiaowen Xin for many contributions including to the PIW workflow, Zhengang Cheng for providing some of the first Web service actors, Werner Krebs for EOL extensions, Steve Mock for Globus actors, Shawn Bowers for his work on semantic types for Kepler, and last but not least, the many scientists and PIs that provide direct or indirect support to this effort, among them Bill Michener, Chaitan Baru, Kim Baldrige, Mark Miller, Arie Shoshani, Terence Critchlow, and Mladen Vouk.

REFERENCES

1. Web Services Description Language (WSDL) Version 1.2, June 2003. http://www.w3.org/TR/wsdl12.
2. OWL Web Ontology Language Reference, W3C Proposed Recommendation, December 2003. http://www.w3.org/TR/owl-ref/.
3. *Scientific Data Management Framework Workshop*, Argonne National Labs, August 2003. Available at: http://sdm.lbl.gov/~arie/sdm/SDM.Framework.wshp.htm.
4. *e-Science Workflow Services Workshop*, e-Science Institute, Edinburgh, U.K., December 2003. Available at: http://www.nesc.ac.uk/esi/events/303/index.html.
5. *e-Science Grid Environments Workshop*, e-Science Institute, Edinburgh, U.K., May 2004. Available at: http://www.nesc.ac.uk/esi/events/.
6. *GRIST Workshop on Service Composition for Data Exploration in the Virtual Observatory*, California Institute of Technology, July 2004. Available at: http://grist.caltech.edu/sc4devo/.
7. *LINK-Up Workshop on Scientific Workflows*, San Diego Supercomputer Center, October 2004. Available at: http://kbis.sdsc.edu/events/link-up-11-04/.
8. *Workflow in Grid Systems Workshop*, GGF10, Berlin, Germany, March 2004. Available at: http://www.extreme.indiana.edu/groc/Worflow-call.html.
9. NSF/ITR. Enabling the Science Environment for Ecological Knowledge (SEEK). http://www.seek.ecoinformatics.org.
10. Michener WK, Beach JH, Jones MB, Ludäscher B, Pennington DD, Pereira RS, Rajasekar A, Schildhauer M. A knowledge environment for the biodiversity and ecological sciences. *Journal of Intelligent Information Systems* 2004.
11. Altintas I *et al.* A modeling and execution environment for distributed scientific workflows. *Proceedings of the 15th International Conference on Scientific and Statistical Database Management (SSDBM)*, Boston, MA, 2003.
12. Bowers S, Ludäscher B. An ontology driven framework for data transformation in scientific workflows. *Proceedings of the International Workshop on Data Integration in the Life Sciences (DILS)*, Leipzig, Germany, March 2004 (*Lecture Notes in Computer Science*, vol. 2994). Springer: Berlin, 2004.
13. Bowers S, Ludäscher B. Actor-oriented design of scientific workflows. *Proceedings of the 24th International Conference on Conceptual Modeling*, Klagenfurt, Austria, October 2005 (*Lecture Notes in Computer Science*). Springer: Berlin (to appear).
14. Kepler: A system for scientific workflows. http://www.kepler-project.org.
15. Werner T. Target gene identification from expression array data by promoter analysis. *Biomolecular Engineering* 2001; **17**:87–94.
16. Peterson L, Yin E, Nelson D, Altintas I, Ludäscher B, Critchlow T, Wyrobek AJ, Coleman MA. Mining the frequency distribution of transcription factor binding sites of ionizing radiation responsive genes. *New Horizons in Genomics, DOE/SC-0071*, Santa Fe, NM, 30 March–1 April 2003.
17. Liu L, Pu C, Han W. An XML-enabled data extraction tool for Web sources. *International Journal of Information Systems (Special Issue on Data Extraction, Cleaning, and Reconciliation)* 2001.
18. National Center for Biotechnology Information (NCBI), 2004. http://www.ncbi.nlm.nih.gov/.
19. Ptolemy II project and system. Department of EECS, UC Berkeley, 2004. http://ptolemy.eecs.berkeley.edu/ptolemyII/.
20. Brooks C, Lee EA, Liu X, Neuendorffer S, Zhao Y, Zheng H. Heterogeneous concurrent modeling and design in Java (vols. 1–3). *Technical Memoranda UCB/ERL M04/27, M04/16, M04/17*, Department of EECS, University of California, Berkeley, 2004.
21. Schmidt M *et al.* The general atomic and molecular electronic structure system. *Journal of Computational Chemistry* 1993; **14**:1347–1363. Available at: http://www.msg.ameslab.gov/GAMESS/GAMESS.html.
22. Abramson D, Giddy J, Kotler L. High performance parametric modeling with Nimrod/G: Killer application for the global Grid. *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, Cancun, Mexico, May 2000. Available at: http://www.csse.monash.edu.au/~davida/nimrod/.
23. Gao J, Thompson MA. (eds.). *Combined Quantum Mechanical and Molecular Mechanical Methods*. American Chemical Society, 1998.
24. Sudholt W, Baldridge K, Abramson D, Enticott C, Garic S. Parameter scan of an effective group difference pseudopotential using Grid computing. *New Generation Computing* 2004; **22**:137–146.
25. NSF/ITR. GEON: A research project to create cyberinfrastructure for the geosciences. http://www.geongrid.org.
26. Scientific Data Management Center (SDM). http://sdm.lbl.gov/sdmcenter/; http://www.npaci.edu/online/v5.17/scidac.html.
27. Biomedical Informatics Research Network Coordinating Center (BIRN-CC), University of California, San Diego, CA. http://nbirn.net/.
28. ROADNet: Real-time observatories, applications and data management network. http://roadnet.ucsd.edu.
29. Sheth A. Changing focus on interoperability in information systems: From system, syntax, structure to semantics. *Interoperating Geographic Information Systems*, Goodchild M, Egenhofer M, Fegeas R, Kottman C (eds.). Kluwer: Dordrecht, 1998; 5–30.

30. Halevy A. Answering queries using views: A survey. *VLDB Journal* 2001; **10**(4):270–294.
31. Nash A, Ludäscher B. Processing unions of conjunctive queries with negation under limited access patterns. *Proceedings of the 9th International Conference on Extending Database Technology (EDBT)*, Heraklion, Crete, Greece, 2004 (*Lecture Notes in Computer Science*, vol. 2992). Springer: Berlin, 2004; 422–440.
32. Ludäscher B, Gupta A, Martone ME. A model-based mediator system for scientific data management. *Bioinformatics: Managing Scientific Data*, Lacroix Z, Critchlow T (eds.). Morgan Kaufmann: San Francisco, CA, 2003.
33. Bowers S, Lin K, Ludäscher B. On integrating scientific resources through semantic registration. *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM)*, Santorini Island, Greece, 2004.
34. *Proceedings of the ICAPS Workshop on Planning for Web Services*, Trento, Italy, June 2003.
35. Blythe J, Deelman E, Gil Y. Planning for workflow construction and maintenance on the Grid. *Proceedings of the ICAPS Workshop on Planning for Web Services*, Trento, Italy, June 2003.
36. Ludäscher B, Altintas I, Gupta A. Compiling abstract scientific workflows into Web service workflows. *Proceedings of the 15th International Conference on Scientific and Statistical Database Management (SSDBM)*, Boston, MA, 2003. Available at: http://kbis.sdsc.edu/SciDAC-SDM/ludaescher-compiling.pdf.
37. Ludäscher B, Nash A. Web service composition through declarative queries: The case of conjunctive queries with union and negation. *Proceedings of the 20th International Conference on Data Engineering (ICDE)*, 2004.
38. Alonso G, Mohan C. Workflow management systems: The next generation of distributed processing tools. *Advanced Transaction Models and Architectures*, Jajodia S, Kerschberg L (eds.), 1997.
39. zur Muehlen M. *Workflow-based Process Controlling*. Logos Verlag: Berlin, 2004.
40. van der Aalst W, van Hee K. *Workflow Management: Models, Methods, and Systems (Cooperative Information Systems)*. MIT Press: Cambridge, MA, 2002.
41. van der Aalst W. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems. Web Services—Been There Done That? Trends and Controversies*, January/February 2003. Available at: http://tmitwww.tm.tue.nl/research/patterns/download/ieeewebflow.pdf.
42. Curbera F, Goland Y, Klein J, Leyman F, Roller D, Thatte S, Weerawarana S. Business Process Execution Language for Web Services (BPEL4WS), Version 1.0, 2002. http://www.ibm.com/developerworks/library/ws-bpel/.
43. van der Aalst W, ter Hofstede A, Kiepuszewski B, Barros A. Workflow patterns. *Distributed and Parallel Databases* 2003; **14**(3):5–51.
44. The Taverna Project. http://taverna.sf.net/.
45. The Triana Project. http://www.trianacode.org/.
46. Chen I, Markowitz V. The Object-Protocol Model: Design, implementation, and scientific applications. *ACM Transactions on Information Systems* 1995; **20**(5).
47. Meidanis J, Vossen G, Weske M. Using workflow management in DNA sequencing. *Proceedings of the International Conference on Cooperative Information Systems (CoopIS)*, 1996.
48. Ailamaki A, Ioannidis YE, Livny M. Scientific workflow management by database management. *Proceedings of the 10th International Conference on Scientific and Statistical Database Management (SSDBM)*, Capri, Italy, 1998.
49. Kiepuszewski B. Expressiveness and suitability of languages for control flow modelling in workflows. *PhD Thesis*, Queensland University of Technology, 2002.
50. Kahn G, MacQueen DB. Coroutines and networks of parallel processes. *Proceedings of the IFIP Congress 77*, Gilchrist B (ed.), 1977; 993–998.
51. Lee EA, Parks T. Dataflow process networks. *Proceedings of the IEEE* 1995; **83**(5):773–799. Available at: http://citeseer.nj.nec.com/455847.html.
52. Morrison JP. *Flow-Based Programming—A New Approach to Application Development*. Van Nostrand Reinhold: New York, 1994.
53. Wright H, Brodlie K, Brown M. The dataflow visualization pipeline as a problem solving environment. *Virtual Environments and Scientific Visualization*, Göbel M, David J, Slavik P, van Wijk JJ (eds.). Springer: Berlin, 1996; 267–276.
54. McPhillips TM. Pipelined scientific workflows for inferring evolutionary relationships. *Unpublished Paper*, Natural Diversity Discovery Project, 2005.
55. Project G. GridFTP—Universal Data Transfer for the Grid, 2000. http://www.globus.org/datagrid/gridftp.html.
56. The Globus Alliance. http://www.globus.org.
57. SDSC Storage Resource Broker. http://www.sdsc.edu/srb/.
58. R—Statistical data analysis. http://www.r-project.org.
59. Eker J, Janneck JW, Lee EA, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE* 2003; **91**(1).
60. Ludäscher B, Altintas I. On providing declarative design and programming constructs for scientific workflows based on process networks. *Technical Report SciDAC-SPA-TN-2003-01*, San Diego Supercomputer Center, 2003. Available at: http://kbi.sdsc.edu/SciDAC-SDM/scidac-tn-map-constructs.

61. Reekie HJ. Realtime signal processing: Dataflow, visual, and functional programming. *PhD Thesis*, School of Electrical Engineering, University of Technology, Sydney, 1995.
62. Nikhil RS, Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann: San Francisco, CA, 2001.
63. Ludäscher B. Towards actor-oriented Web service-based scientific workflows (or: How to handle handles). *Technical Report*, San Diego Supercomputer Center, September 2004.
64. Weinstein D, Parker S, Simpson J, Zimmerman K, Jones G. Visualization in the SCIRun problem-solving environment. *Visualization Handbook*, Hansen C, Johnson C (eds.). Elsevier: Amsterdam, 2005; 615–632.
65. Oinn T *et al.* Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics Journal* 2004; **20**(17):3045–3054.
66. Churches D, Gombas G, Harrison A, Maassen J, Robinson C, Shields M, Taylor I, Wang I. Programming scientific and distributed workflow with Triana services. *Concurrency and Computation: Practice and Experience* 2006; [this issue].
67. Deelman E *et al.* Mapping abstract complex workflows onto Grid environments. *Journal of Grid Computing* 2003; **1**(1):25–39.
68. Thain D, Tannenbaum T, Livny M. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience* 2005; **17**(2–4):323–356.
69. Yu J, Buyya R. A taxonomy of workflow management systems for Grid computing. *Technical Report GRIDS-TR-2005-1*, Grid Computing and Distributed Systems Laboratory, University of Melbourne, 2005. Available at: http://www.gridbus.org/reports/GridWorkflowTaxonomy.pdf.
70. Takatuska M, Gahegan M. GeoVISTA Studio: A codeless visual programming environment for geoscientific data analysis and visualization. *Computers and Geosciences* 2002; **28**(2):1131–1144.
71. Armstrong R, Gannon D, Geist A, Keahey K, Kohn S, McInnes L, Parker S, Smolinski B. Toward a common component architecture for high-performance scientific computing. *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation*, August 1999.

